Singleton, Command, & Adaptor

Kenneth M. Anderson
University of Colorado, Boulder
CSCI 4448/6448 — Lecture 21 — 11/04/2008

VOTE!

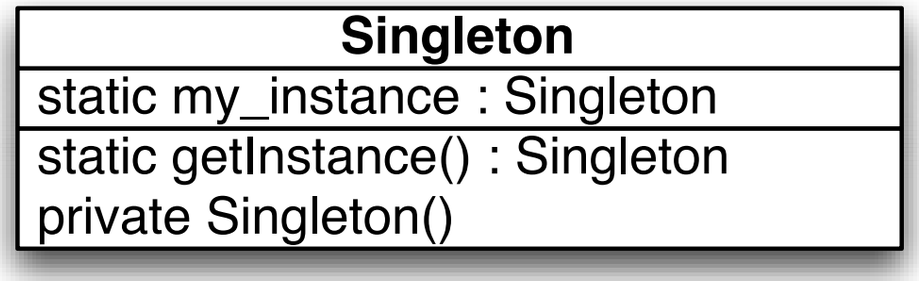
Lecture Goals

- Cover Material from Chapters 5 — 7 of the Design Patterns Textbook
    - Singleton Pattern
    - Command Pattern
    - Adaptor Pattern
    - Facade Pattern

Singleton Pattern: Definition

- The Singleton Pattern ensures a class has only one instance (or a constrained set of instances), and provides a global point of access to it
  - Useful for objects that represent real-world resources, such as printers, in which you want to instantiate one and only one object to represent each resource
    - Also useful for “management” code, such as a thread/connection pool
- At first, Singleton may seem difficult to achieve… typically, once you define a class, you can create as many instances as you want
  - Foo f = new Foo(); Foo f1 = new Foo(); Foo f2 = new Foo()…
- The key (in most languages) is to limit access to the class’s constructor, such that only code in the class can invoke a call to the constructor (or initializer or <insert code that creates instances here>)
  - Indeed, as you will see, different languages achieve the Singleton pattern in different ways

Singleton Pattern: Structure



Singleton involves only a single class (not typically called Singleton). That class is a full-fledged class with other attributes and methods (not shown)

The class has a static variable that points at a single instance of the class.

The class has a private constructor (to prevent other code from instantiating the class) and a static method that provides access to the single instance

World's Smallest Java-based Singleton Class

```
1  public class Singleton {
2
3          private static Singleton uniqueInstance;
4
5          private Singleton() {}
6
7          public static Singleton getInstance() {
8                  if (uniqueInstance == null) {
9                          uniqueInstance = new Singleton();
10                 }
11                 return uniqueInstance;
12         }
13 }
14
```

Meets Requirements: static var, static method, private constructor

Example source has this class in ken/smallest augmented with test code.

World's Smallest Python-Based Singleton Class

```
1  class Singleton(object):
2
3      _instance = None
4
5      def __new__(cls, *args, **kwargs):
6          if not cls._instance:
7              cls._instance = super(Singleton, cls).__new__(cls, *args, **kwargs)
8          return cls._instance
9
10 if __name__ == '__main__':
11     a = Singleton()
12     b = Singleton()
13
14     print "a = %s" % (a)
15     print "b = %s" % (b)
16
```

Different Approach: static var, override constructor

only 8 lines of code!

Example source has this class in ken/smallest

World's Smallest Ruby-based Singleton Class

```
1  require 'singleton'
2
3  class Example
4    include Singleton
5  end
6
7  a = Example.instance
8  b = Example.instance
9
10 puts "a = #{a}"
11 puts "b = #{b}"
12
13 c = Example.new
14
```

Yet a different approach, using a mechanism in Ruby called a "mixin"

The "include Singleton" statement causes the Example class to be modified such that its new() method becomes private and an instance() method is added to retrieve an instance. As a bonus, it will also handle hiding allocate(), overriding the clone() and dup() methods, and is thread safe!

Only 5 lines of code!

Thread Safe?

- The Java and Python code just shown is not thread safe
  - This means that it is possible for two threads to attempt to create the singleton for the first time simultaneously
  - If both threads check to see if the static variable is empty at the same time, they will both proceed to creating an instance and you will end up with two instances of the singleton object (not good!)
  - Example Next Slide

Program to Test Thread Safety

```
1  public class Creator implements Runnable {
2
3      private int id;
4
5      public Creator(int id) {
6          this.id = id;
7      }
8
9      public void run() {
10         try {
11             Thread.sleep(200L);
12         } catch (Exception e) {
13         }
14         Singleton s = Singleton.getInstance();
15         System.out.println("s" + id + " = " + s);
16     }
17
18     public static void main(String[] args) {
19         Thread[] creators = new Thread[10];
20         for (int i = 0; i < 10; i++) {
21             creators[i] = new Thread(new Creator(i));
22         }
23         for (int i = 0; i < 10; i++) {
24             creators[i].start();
25         }
26     }
27
28 }
29
```

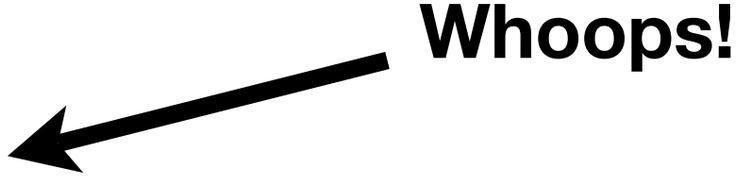
Creates a “runnable” object that can be assigned to a thread.

When its run, its sleeps for a short time, gets an instance of the Singleton, and prints out its object id.

The main routine, creates ten runnable objects, assigns them to ten threads and starts each of the threads

Output for Non Thread-Safe Singleton Code

- s9 = Singleton@45d068
- s8 = Singleton@45d068
- s3 = Singleton@45d068
- s6 = Singleton@45d068
- s1 = Singleton@45d068
- s0 = Singleton@ab50cd
- s5 = Singleton@45d068
- s4 = Singleton@45d068
- s7 = Singleton@45d068
- s2 = Singleton@45d068



Thread 0 created an instance of the Singleton class at memory location ab50cd at the same time that another thread (we don't know which one) created an additional instance of Singleton at memory location 45d068!

How to Fix?

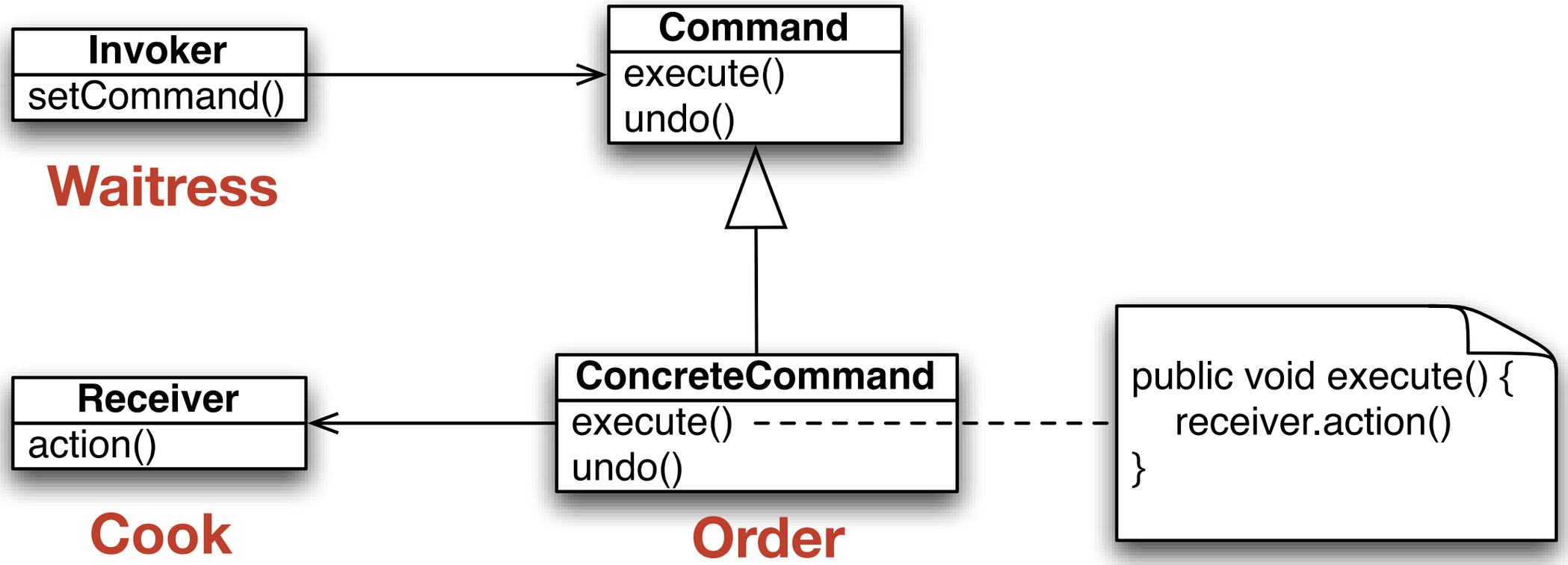
```
1  public class Singleton {
2
3          private static Singleton uniqueInstance;
4
5          private Singleton() {}
6
7          public static synchronized Singleton getInstance() {
8                  if (uniqueInstance == null) {
9                          uniqueInstance = new Singleton();
10                 }
11                 return uniqueInstance;
12         }
13
14 }
15
```

In Java, the easiest fix is to add the **synchronized** keyword to the getInstance() method. The book talks about other methods that address performance-related issues. My advice: use this approach first!

Command Pattern: Definition

- The Command Pattern encapsulates a request as an object, thereby letting you parameterize other objects with different requests, queue or log requests, and support undoable operations
- Think of a Restaurant
  - You, the Customer, give your Waitress an Order
  - The Waitress takes the Order to the kitchen and says “Order Up”
  - The Cook takes the Order and prepares your meal
    - Think of the order as making calls on the Cook like “makeBurger()”
- A request (Order) is given to one object (Waitress) but invoked on another (Cook)
  - This decouples the object making the request (Customer) from the object that responds to the request (Cook); This is good if there are potentially many objects that can respond to requests

Command Pattern: Structure



I'm leaving one piece out of this diagram: the client.

In order for this pattern to work, someone needs to create a command object and set its receiver. And, someone needs to give command objects to an invoker to invoke at a later time.

Those "someones" may be the same object, they may be different objects

Example: Remote Control

- The example in the textbook involves a remote control for various household devices.
  - Each device has a different interface (plays role of Receiver)
  - Remote control has uniform interface (plays role of Client): “on” and “off”
  - Command objects are created to “load” into the various slots of the remote control
    - Each command has an execute() method that allows it to emit a sequence of commands to its associated receiver
      - Light: turn light on
      - Stereo: turn Stereo on, select “CD”, play()
- In this way, the details of each receiver are hidden from the client. The client simply says “on()” which translates to “execute()” which translates to the sequence of commands on the receiver: nice loosely-coupled system

Enabling Undo

- The command pattern is an excellent mechanism for enabling undo functionality in your application designs
  - The execute() method of a command performs a sequence of actions
  - The undo() method performs the reverse sequence of actions
- Assumption: undo() is being invoked right after execute()
  - If that assumption holds, the undo() command will return the system to the state it was in before the execute() method was invoked
- Since the Command class is a full-fledged object, it can track “previous values” of the system, in order to perform the undo() request
  - Example in book of a command to control “fan speed”. Before execute() changes the speed, it records the previous speed in an instance variable

Macro Commands

- Another nice aspect of the Command pattern is that it is easy to create Macro commands.
  - You simply create a command that contains an array of commands that need to be executed in a particular order
  - execute() on the macro command, loops through the array of commands invoking their execute() methods
  - undo() can be performed by looping through the array of commands backwards invoking their undo() methods
- From the standpoint of the client, a Macro command is simply a “decorator” that shares the same interface as normal Command objects
  - This is an example of one pattern building on another

Demonstration

- The example code for this lecture demonstrates several aspects of the Command pattern
  - Simple commands
  - Simple Undo
  - Macro Commands

Additional Uses: Queuing

- The command pattern can be used to handle the situation where there are a number of jobs to be executed but only limited resources available to do the computations
    - Make each job a Command
    - Put them on a Queue
    - Have a thread pool of computation threads
    - And one thread that pulls jobs off the queue and assigns them to threads in the thread pool
        - If all computation threads are occupied, then the job manager thread blocks and waits for one to become free

Additional Uses: Logging

- This variation involves adding store() and load() methods to command objects that allow them to be written and read to and from a persistent store
    - The idea is to use Command objects to support system recovery functionality
- Imagine a system that periodically saves a “checkpoint” of its state to disk
    - Between checkpoints, it executes commands and saves them to disk
    - Imagine the system crashes
    - On reboot, the system loads its most recent “checkpoint” and then looks to see if there are saved commands
        - If so, it executes those commands in order, taking the system back to the state it was in just before the crash

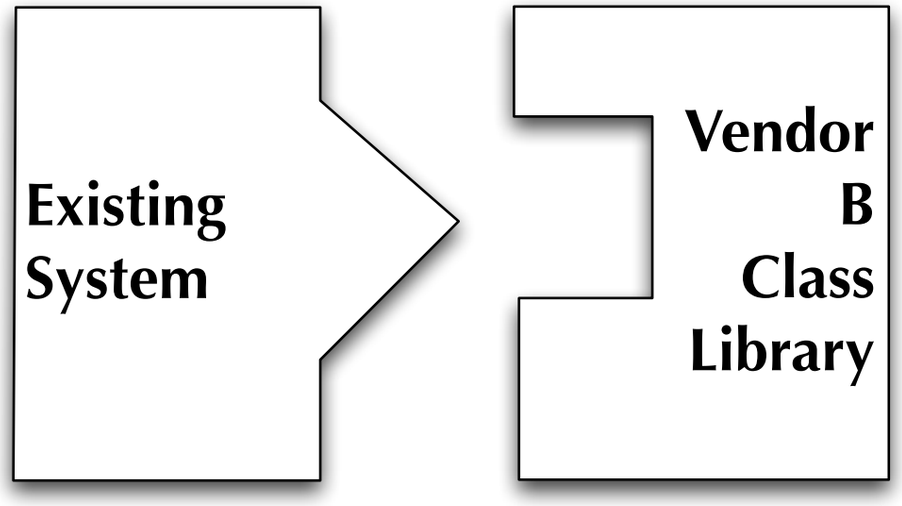
Adapters in the Real World

- Our next pattern provides techniques for converting an interface that is not compatible with an existing system into a different interface that is
  - Real World Example: AC Power Adapters
  - Electronic products made for the USA cannot be used directly with electrical outlets found in most other parts of the world
    - US 3-prong (grounded) plugs are not compatible with European wall outlets
      - To use, you need either
        - an AC power adapter, if the US product has a “universal” power supply, or
        - an AC power convertor/adapter, if it doesn’t
- By example, OO adapters may simply provide adaptation services from one interface to another, or may require more smarts to convert information from one interface before passing it to the second interface

OO Adapters (I)

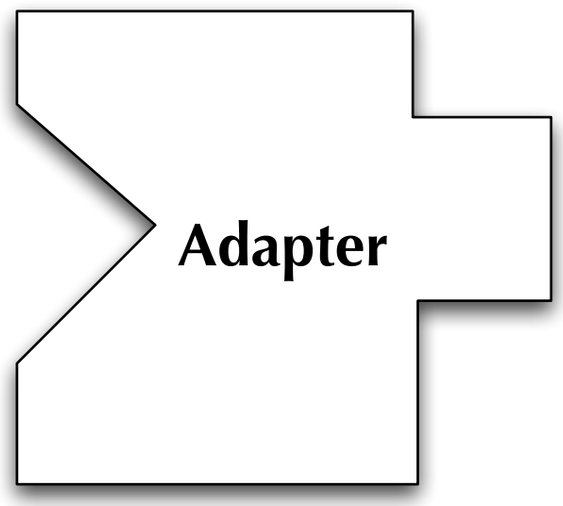
- Pre-Condition: You are maintaining an existing system that makes use of a third-party class library from vendor A
- Stimulus: Vendor A goes belly up and corporate policy does not allow you to make use of an unsupported class library.
- Response: Vendor B provides a similar class library but its interface is completely different from the interface provided by vendor A
- Assumptions: You don’t want to change your code, and you can’t change vendor B’s code.
- Solution?: Write new code that adapts vendor B’s interface to the interface expected by your original code

OO Adapters (II)



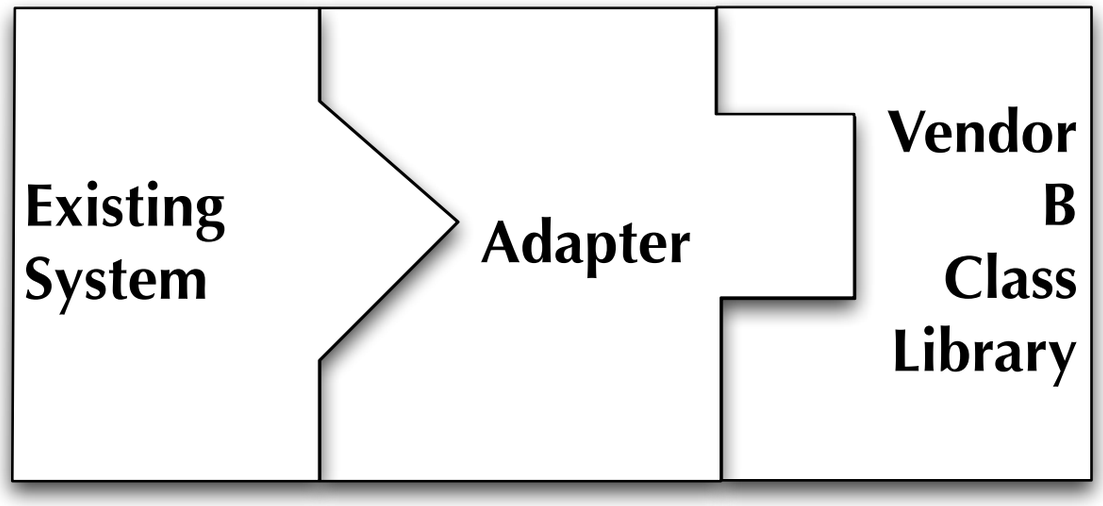
Interface Mismatch
Need Adapter

Create Adapter



And then...

OO Adapters (III)



...plug it in

Benefit: Existing system and new vendor library do not change, new code is isolated within the adapter.

Simple Example: A turkey hiding among ducks! (I)

- If it walks like a duck and quacks like a duck, then it must be a duck!

Simple Example: A turkey hiding among ducks! (II)

- If it walks like a duck and quacks like a duck, then it ~~must~~ **might** be a ~~duck~~ **turkey wrapped with a duck adapter**… (!)
- Recall the Duck simulator from chapter 1?

```
1  public interface Duck {
2      public void quack();
3      public void fly();
4  }
5
6  public class MallardDuck implements Duck {
7
8      public void quack() {
9          System.out.println("Quack");
10     }
11
12     public void fly() {
13         System.out.println("I'm flying");
14     }
15 }
16
```

Simple Example: A turkey hiding among ducks! (III)

- An interloper wants to invade the simulator

```
1  public interface Turkey {
2      public void gobble();
3      public void fly();
4  }
5
6  public class WildTurkey implements Turkey {
7
8      public void gobble() {
9          System.out.println("Gobble Gobble");
10     }
11
12     public void fly() {
13         System.out.println("I'm flying a short distance");
14     }
15
16 }
17
```

Simple Example: A turkey hiding among ducks! (IV)

- Write an adapter, that makes a turkey look like a duck

```
1  public class TurkeyAdapter implements Duck {
2
3      private Turkey turkey;
4
5      public TurkeyAdapter(Turkey turkey) {
6          this.turkey = turkey;
7      }
8
9      public void quack() {
10         turkey.gobble();
11     }
12
13     public void fly() {
14         for (int i = 0; i < 5; i++) {
15             turkey.fly();
16         }
17     }
18
19 }
20
```

Demonstration

1. Adapter implements target interface (Duck).

2. Adaptee (turkey) is passed via constructor and stored internally

3. Calls by client code are delegated to the appropriate methods in the adaptee

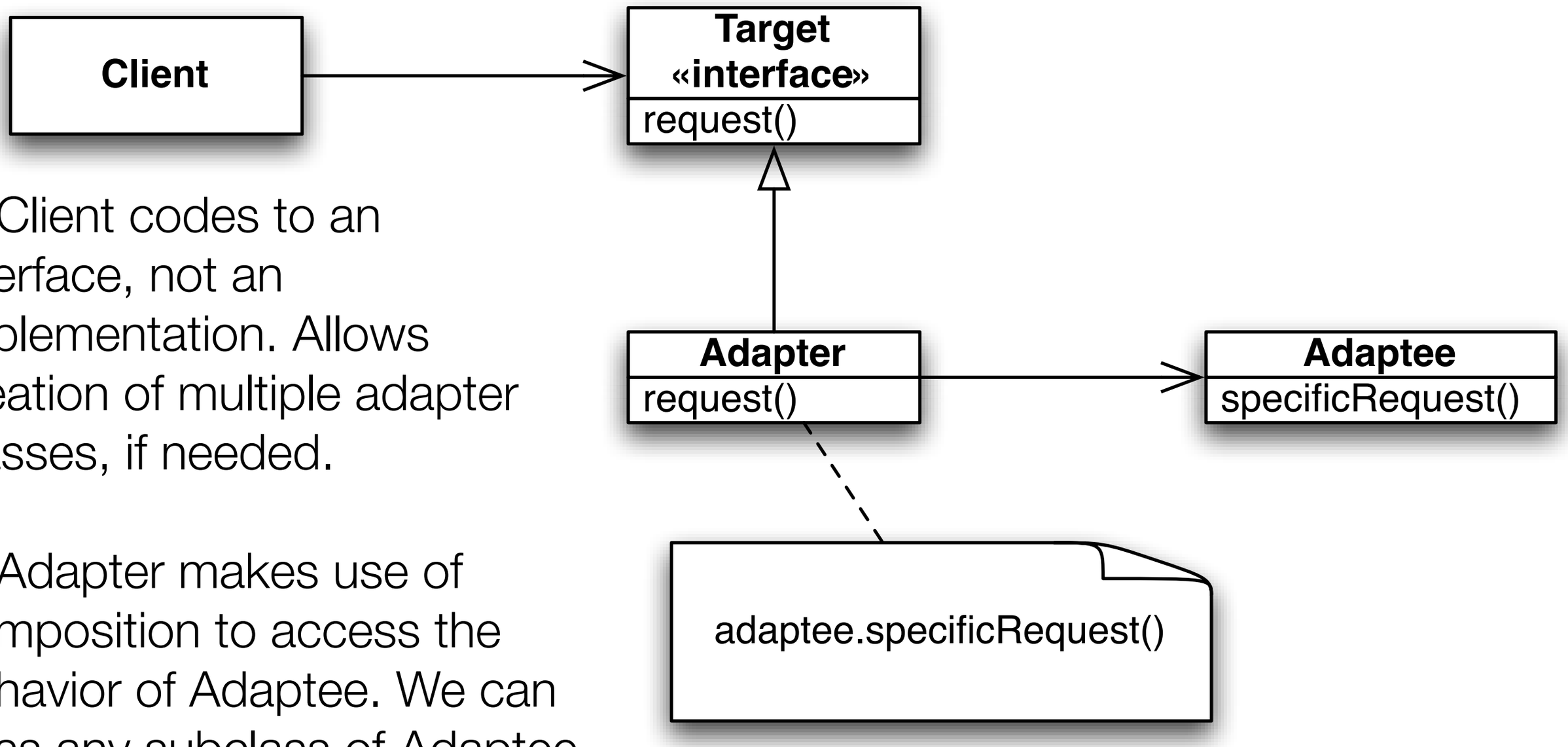
4. Adapter is full-fledged class, could contain additional vars and methods to get its job done

Adapter Pattern: Definition

- The Adapter pattern converts the interface of a class into another interface that clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces
    - The client makes a request on the adapter by invoking a method from the target interface on it
    - The adapter translates that request into one or more calls on the adaptee using the adaptee interface
    - The client receives the results of the call and never knows there is an adapter doing the translation

Adapter Pattern: Structure (I)

Object Adapter

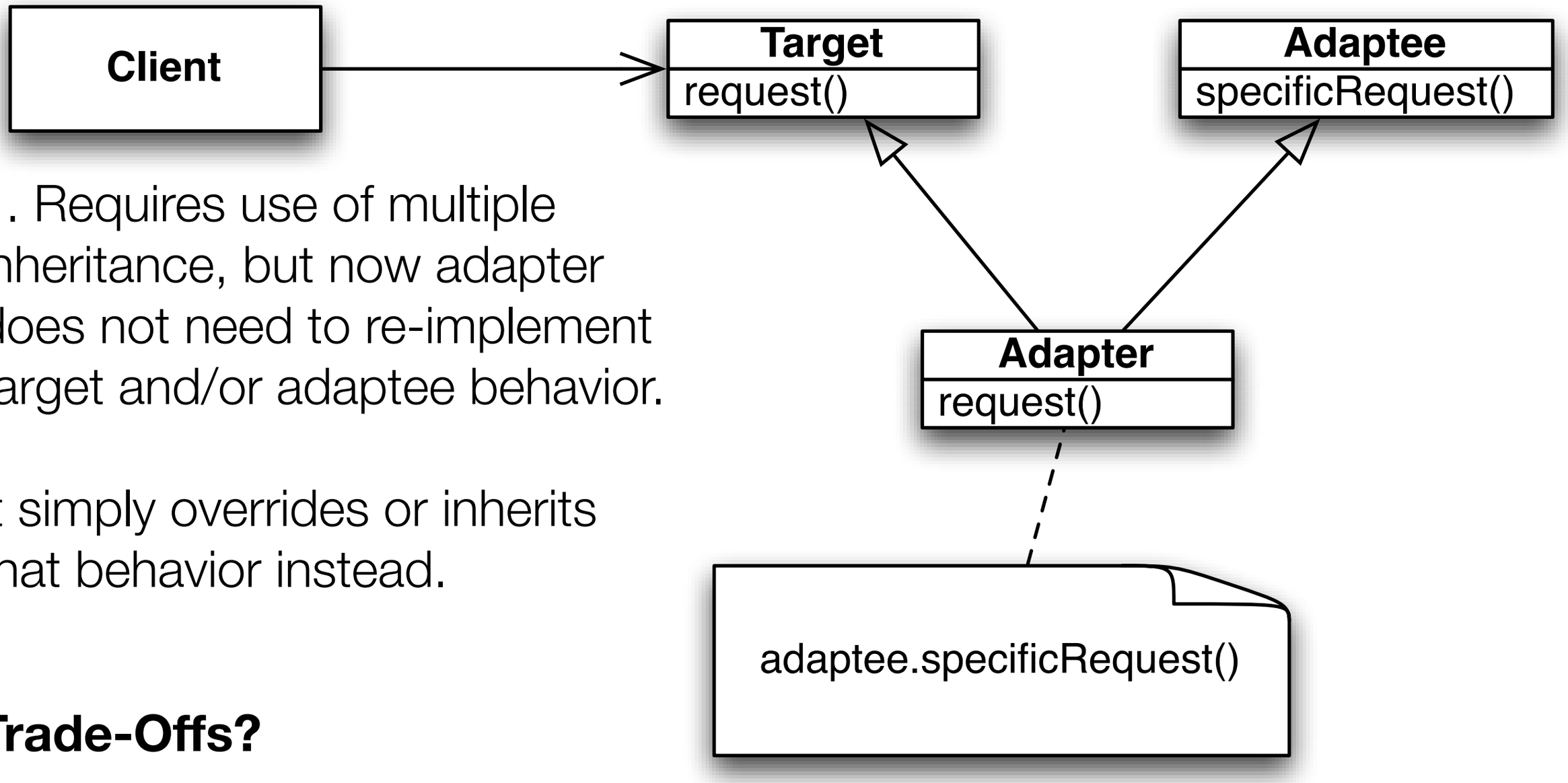


1. Client codes to an interface, not an implementation. Allows creation of multiple adapter classes, if needed.

2. Adapter makes use of composition to access the behavior of Adaptee. We can pass any subclass of Adaptee to the Adapter, if needed.

Adaptee Pattern: Structure (II)

Class Adapter



1. Requires use of multiple inheritance, but now adapter does not need to re-implement target and/or adaptee behavior.

It simply overrides or inherits that behavior instead.

Trade-Offs?

Demonstration

Real World Adapters

- Before Java's new collection classes, iteration over a collection occurred via java.util.Enumeration
  - hasMoreElements() : boolean
  - nextElement() : Object
- With the collection classes, iteration was moved to a new interface: java.util.Iterator
  - hasNext(): boolean
  - next(): Object
  - remove(): void
- There's a lot of code out there that makes use of the Enumeration interface
  - New code can still make use of that code by creating an adapter that converts from the Enumeration interface to the Iteration interface
    - Demonstration

Difference between Adapter and Decorator

- Adapter and Decorator's seem similar: how so?
- Answers
  - They both wrap objects at run-time
  - They both delegate requests to their wrapped objects
- How are they different?
- Answers
  - Adapter converts one interface into another while maintaining functionality
  - Decorator leaves the interface alone but adds new functionality
    - Decorators are designed to be "stacked"; that's less likely to occur with adapters

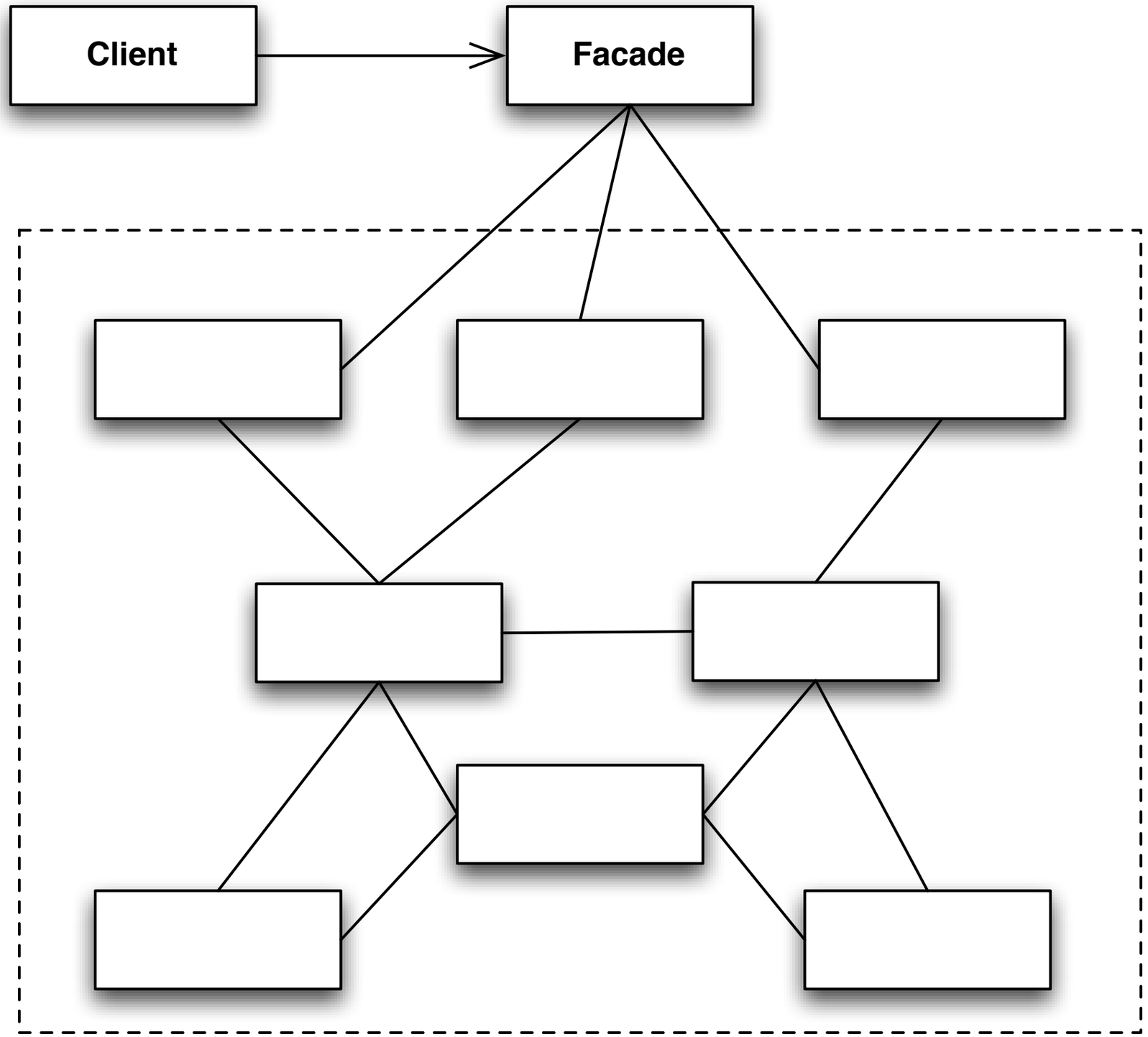
Yet Another Adapter: Facade Pattern

- There is another way in which an adapter can be used between a client and an adaptee: to simplify the interface of the adaptee(s)
- Imagine a library of classes with a complex interface and/or complex interrelationships
  - Book's Example: Home Theater System
    - Amplifier, DvdPlayer, Projector, CdPlayer, Tuner, Screen, PopcornPopper (!), and TheatreLights
      - each with its own interface and interclass dependencies
  - Imagine steps for "watch movie"
    - turn on popper, make popcorn, dim lights, screen down, projector on, set projector to DVD, amplifier on, set amplifier to DVD, DVD on, etc.
  - Now imagine resetting everything after the movie is done, or configuring the system to play a CD, or play a video game, etc.

Facade Pattern: Definition

- The Facade Pattern provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
    - We place high level methods like “watch movie”, “reset system”, “play cd” in a facade object and encode all of the steps for each high level service in the facade.
    - Client code is simplified and the client’s dependencies are greatly reduced
        - A facade not only simplifies an interface, it decouples a client from a subsystem of components
- Relationship to Adapter Pattern?
    - Both facades and adapters may wrap multiple classes, but a facade’s intent is to simplify, while an adapter’s is to convert between interfaces

Facade Pattern: Structure



Demonstration

New Design Principle

- The facade pattern demonstrates a new design principle
- **Principle of Least Knowledge**: “Talk only to your immediate friends”
    - reminds you to create loosely coupled systems of cohesive objects
    - also known as **The Law of Demeter**

- We want to reduce an object’s class dependencies to the bare minimum
- How many classes is this code coupled to?

```
public float getTemp() {
    return station.getThermometer().getTemperature();
};
```

Principle of Least Knowledge: Heuristics

- In order to implement the principle of least knowledge, follow these guidelines
  - For any object
    - Within any method of that object
      - you may invoke methods that belong to
        - the object itself
        - objects passed in as a parameter to the method
        - any object the method creates or instantiates
        - any object that is stored as an instance variable of the host object
- The code on the previous slide violates these guidelines because we invoke the method getTemperature() on a “friend of a friend”
  - Change code to “return station.getTemperature()” to follow guidelines
  - Requires adding “wrapper” method to station class

Example of all the “legal” method invocations

```
1  public class Car {
2
3      private Engine engine;
4
5      public Car() {
6      }
7
8      public void start(Key key) {
9
10         Door doors = new Doors();
11
12         boolean authorized = key.turns();
13
14         if (authorized) {
15             engine.start();
16             updateDashboardDisplay();
17             doors.lock();
18         }
19     }
20
21     public void updateDashboardDisplay() {
22     }
23
24 }
25
```

object passed as parameter

component method

local method

object created by method

Wrapping Up

- Singleton allows you to manage the number of instances a class can have
- Command allows you to separate the “client” and “server” of a method call
- Adapter allows you to convert one interface into another, allowing the client code and the adaptee to remain unchanged
- Decorator seen in new light: an adapter that “converts” an interface into itself while adding new behaviors
- Facade is a variant of the adapter pattern in which the purpose is to (greatly) simplify the adaptee’s interface
- Facade demonstrates the use of a new design principle, the Principle of Least Knowledge, also known as the Law of Demeter
  - often phrased “talk only to your friends”
  - focus is on reducing coupling between classes

Coming Up Next

- Lecture 22: Template Methods, Iterator & Composite
  - Read Chapters 8 — 9 of the Design Patterns Textbook